Structured Programming (1991) 12: 17-30

Structured Programming
©1991 Springer-Verlag New York Inc.

Stepwise Refinement of Action Systems

Ralph-Johan Back, Kaisa Sere

Åbo Akademi University, Department of Computer Science, Lemminkäisenkatu 14, SF-20520 Turku, Finland
e-mail: backrj@finabo.abo.fi

Abstract. A method for the formal development of provably correct parallel algorithms by stepwise refinement is presented. The entire derivation procedure is carried out in the context of purely sequential programs. The resulting parallel algorithms can be efficiently executed on different architectures. The methodology is illustrated by showing the main derivation steps in a construction of a parallel algorithm for matrix multiplication.

Key Words: action systems, distributed action systems, parallel algorithms, provability, refinement calculus, stepwise refinement

1. Introduction

Stepwise refinement is one of the main methods for systematic construction of sequential programs: a high level specification of a program is transformed by a sequence of correctness preserving transformations into an executable and efficient program that satisfies the original specification. The *refinement calculus* is a formalization of the stepwise refinement approach. It was first described in [1, 2] and has been further elaborated in [3, 17, 18].

The *action system* formalism for parallel and distributed computations was introduced in [6] and is further developed in, e.g., [7]. The behaviour of parallel and distributed programs is described in terms of the actions which processes in the system carry out in co-operating with each other. Several actions can be executed in parallel, as long as the actions do not have any variables in common. The actions are atomic: if an action is chosen for execution, it is executed to completion without any interference from the other actions in the system.

Atomicity guarantees that a parallel execution of an action system gives the same results as a sequential and nondeterministic execution. This allows us to use the sequential refinement calculus to construct parallel action systems by stepwise refinements. We can start our derivation from a more or less sequential algorithm and successively increase the degree of parallelism in it, while preserving the correctness of the algorithm. Parallelism is introduced by merging action systems and refining the atomicity of actions [4].

The refinement calculus is based on the assumption that the notion of correctness we want to preserve is total correctness. The refinement relation is not bound to the choice of this specific notion of correctness, but much of the methods developed and the theory is specific to this choice. Total correctness is the appropriate correctness notion for *parallel algorithms*. These programs differ from sequential algorithms only in that they are executed in parallel, by co-operation of many processes. They are intended to terminate, and only the final results are of interest. The refinement calculus and the action system formalism together provide a powerful and uniform method for deriving parallel algorithms by stepwise refinement.

The action system approach is the topic of Section 2. A classification of action systems based on the characteristics which allow their efficient implementation on different machine architectures is also briefly discussed there. The refinement calculus for statements and actions is presented in Section 3. In Section 4 we describe the methods needed to increase parallelism in action systems. These methods are illustrated by the derivation of a nontrivial parallel algorithm for matrix multiplication in Section 5. We end with some concluding remarks in Section 6.

2. Action Systems

An *action system* $\mathcal{A}$ is a collection of *actions* $\{A_1, \dots, A_m\}$ on some set of *state variables* $x = \{x_1, \dots, x_n\}$. Each variable is associated with some domain of values. The set of possible assignments of values to the state

variables constitutes the *state space* Σ. Each action A_i is of the form $g_i \rightarrow S_i$ where the *guard* g_i is a boolean condition and the *body* S_i a sequential, possibly nondeterministic statement on the state variables. An *initialization statement* S_0 assigns initial values to the variables x.

An action system describes the state space of a system and the possible actions that can be executed in the system. The way in which the actions are executed depends on the evaluation mechanism that we postulate for the system. The simplest evaluation mechanism is *sequential*: the behaviour of the action system $\mathcal{A}$ is that of the guarded iteration statement

$$S_0;\ \mathbf{do}\ A_1\ []\ \ldots\ []\ A_m\ \mathbf{od}$$

on the state variables x [11]; i.e., actions are executed sequentially with nondeterministic choices when more than one action is enabled.

The following is a sorting program for exchange sort, described as an action system:

$[\ \mathit{var}\ x.1, \ldots, x.n\ \mathit{is\ integer};$
$\quad x.1, \ldots, x.n := X.1, \ldots, X.n;$
$\mathbf{do}\ x.1 > x.2 \rightarrow x.1, x.2 := x.2, x.1$
$\ldots$
$[]\ x.(n-1) > x.n \rightarrow x.(n-1), x.n := x.n, x.(n-1)$
$\mathbf{od}]$

This program will sort n integers $X.1, \ldots, X.n$ in ascending order. We can look upon this program as an action system with an initialization statement and $n-1$ sorting actions. The program terminates in a state where the array x is a permutation of the original array X and $x.i \leq x.(i+1)$ for $i = 1, \ldots, n-1$.

Action systems can also be executed in *parallel*. We consider here two different ways of parallel execution: *concurrent action system* and *distributed action system*. In the concurrent execution model the *actions* are partitioned among the processes. This gives us a shared variable model for communication and synchronization. Each action is assigned to some specific process. A variable that is referenced by at least two different actions in two different processes is *shared*, while a variable that is referenced only by actions in one process is *private* to that process. The actions are executed in parallel, with the restriction that all actions are *atomic*: two actions that share a common variable may not execute at the same time.

In the *distributed action system* the *variables* (rather than the actions) are partitioned among the processes. This results in a distributed execution model where all variables are local and processes synchronize and communicate by a generalized handshaking mechanism. Each variable is assigned to a unique process. An action is *shared* if it refers to variables in two or more processes and private if it only refers to variables in one process. A shared action is assumed to be executed jointly by all the processes sharing this action. The processes are therefore synchronized for execution of such an action. Shared actions also provide communication between processes: a variable in one process may be updated in a way that depends on the values of variables in other processes involved in the shared action. The actions are executed in parallel, again with the restriction enforced by atomicity: no two actions involving a common process may execute simultaneously. This implies that no two actions refering to a common variable may execute in parallel.

The example sorting program becomes a concurrent action system if we assign each of the $n-1$ actions to a process of its own. The variables $x.2, \ldots, x.(n-1)$ are then shared by two processes each, while $x.1$ and $x.n$ are private. The program becomes a distributed action system if we assign each element of the array x to a process of its own. Each action in the system is then shared by exactly two processes. In both cases, actions that do not have any variables $x.i$ in common can execute in parallel.

Classification of Distributed Action Systems

The distributed action systems can be divided into subclasses, depending on the way in which they are implementable on different kinds of distributed architectures. Let $\mathcal{A}$ be an action system with initialization S_0, actions $A = \{A_1, \ldots, A_m\}$ and state variables $x = \{x_1, \ldots, x_n\}$. We will identify a *process* with a subset of state variables x. Let $p = \{p_1, \ldots, p_k\}$ be a partitioning of x into processes; i.e., $p_i \subset x$ for each i, $x = \cup p$ and $p_i \cap p_j = \emptyset$ for any $i, j, i \neq j$. The action A_i (or action guard g, action body S) is said to *involve* process p_j if A_i (or g, S) refers to some variable in p_j.

An action A_i is of *degree n* if it involves n processes. An action system is of *degree n* if each of its actions is of at most degree n. The usual message passing models for parallel programming, such as Ada and CSP/Occam, correspond to action systems of degree 2. Synchronous broadcasting among N processes, e.g., to execute the same program on different processes in lock-step, corresponds to a system with degree N. The distributed sorting algorithm is of degree 2.

An action A_i is *decentralized* if its guard g_i is a boolean combination of *primitive guards* $g_i^1, \ldots, g_i^r$, each of which only involves a single process. An action system is *decentralized* if each of its actions is decentralized. The processes can in such a system determine the truth of primitive guards by only inspecting their own local variables. The enabledness of

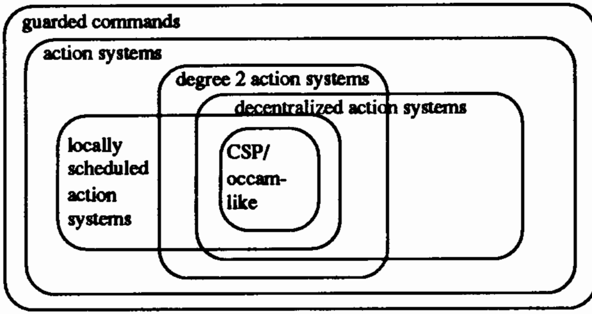

Figure 1. A classification of distributed action systems.

an action is thus determined in a distributed fashion. The actual scheduling of actions for execution may be done in a centralized or distributed fashion.

A process p involved in action A_i is *committed* to the action if, whenever action A_i is enabled, no other action in $\mathcal{A}$ that p is involved in can be enabled. An action A_i is *locally scheduled* in $\mathcal{A}$ if at most one uncommitted process is involved in the action. (The uncommitted action is then called a *scheduler* for the action.) The action system $\mathcal{A}$ is *locally scheduled* if each of its actions is locally scheduled. In such a system, we do not need any expensive agreement protocols to decide which of the possibly conflicting enabled actions should be executed because the scheduler of each action can make this decision for itself, without danger of deadlock. The local scheduling condition is enforced in, e.g., CSP/Occam by disallowing output guards and in Ada by the asymmetric treatment of callers and callees.

The different classes of action systems are illustrated in Figure 1. We see that a programming language such as Occam forms a subset of decentralized and locally scheduled action systems of degree 2. It is only a subset because it only permits an assignment statement as the body of a communication action, with the additional restriction that the variables in one of the processes may only be read and the variables in the other process may only be written.

Action systems generalize the communication mechanisms usually found in programming languages for parallel and distributed programming. Parallel implementations of action systems require some additional mechanism to enforce atomicity and to schedule the execution of actions. In [6] it is shown how decentralized action systems of degree 2 can be implemented in CSP with output guards. Efficient algorithms to implement decentralized action systems of any degree on broadcasting networks are presented in [5, 7]. In [9] we show how a class of decentralized, locally scheduled action systems of any degree can be efficiently implemented in Occam [13], which does not permit output guards.

Programming with Action Systems

The use of action systems permits the design of the logical behaviour of a system to be separated from the issue of how the system is to be implemented. The latter is seen as a design decision that does affect the way in which the action system is built, but is not reflected in the logical behaviour of the system. The decision whether the action system is to be executed in a sequential, concurrent or distributed fashion can be postponed to a later stage, when the logical behaviour of the action system has been designed. The construction of the program is thus done within a single unifying framework.

The stepwise refinement of action systems would start with a specification of the intended behaviour of the system, in the form of a sequential statement. The goal is to construct an action system that fits into one of the classes described above, with a specific target architecture in mind. The required parallel action system is constructed by making small refinements in the original statement, until an action system satisfying the requirements has been constructed.

3. Refinement Calculus

We restrict ourselves to the language of guarded commands [11], with some extensions. We have two different syntactic categories, *statements* and *actions*, which we define as follows. A *statement* S is defined as

$S ::=$	$x := x'.Q$	(*nondeterministic assignment*)
|	$\{Q\}$	(*assert statement*)
|	$S_1; \ldots ; S_n$	(*sequential composition*)
|	$[A_1 [] \ldots [] A_n]$	(*conditional composition*)
|	$*[A_1 [] \ldots [] A_n]$	(*iterative composition*)
|	$[var\ x_1 : T_1; \ldots ; x_n : T_n; S]$	(*block with local variables*)

Here $A_1, \ldots, A_n$ are actions, x and x' are (lists of) variables and Q is a predicate. We write $(i : 1..n : S_i)$ for $S_1; \ldots ; S_n$ and similarly $[i: 1..n: A_i]$, $*[i : 1..n: A_i]$ and $[var\ x.i : T_i, i : 1..n; S]$ for the other constructs.

The *nondeterministic assignment statement* $x := x'.Q$ [2] permits specifications to be treated as statements. It assigns to the variables x some values x' that make the condition Q true. The statement aborts if this is not possible. The *assert statement* $\{Q\}$ acts as *skip* if the condition Q holds in the initial state. If the condition Q does not hold in the initial state, then the effect is the same as *abort*. The other statements have their usual meanings. The *(multiple) assignment statement* $x_1, \ldots, x_n := e_1, \ldots, e_n$ is a special case of the nondeterministic assignment statement, definied as $x_1, \ldots, x_n := x'_1, \ldots, x'_n.(x'_1 = e_1 \wedge \ldots \wedge x'_n = e_1)$. We write it as $\| i{:}1..n{:}\ x_i := e_i \|$.

An *action* (or guarded command) A is of the form

$$A ::= g \to S$$

where g is a boolean condition (the *guard*) and S is a statement (the *body*). We will write gA for the guard of action A and bA for the body of action A.

An *action system* $\mathcal{A}$ is simply a statement of the form

$$[var\ y_i : T_i,\ i : 1..m;\ S_0;\ *[i : 1..n : A_i]]$$

where $A_1, \ldots, A_n$ are actions. The variables $y_1, \ldots y_m$ are local to the action system, while other variables referenced in the system are global.

The *weakest precondition* wp(S, R) for *statement* S to establish postcondition R is definied in the usual way [11], with the adaptions needed for the unbounded nondeterminism introduced by the nondeterministic assignment statement [3]. The weakest precondition for an *action* $A = g \to S$ to establish postcondition R is definied as

$$\mathrm{wp}(g \to S, R) = (g \Rightarrow \mathrm{wp}(S, R)).$$

The weakest precondition for a finite sequence $A_1; A_2; \ldots A_k$ of actions to establish postcondition R is definied in the same way as for statements,

$$\mathrm{wp}(A_1; A_2; \ldots A_k, R) =$$
$$\mathrm{wp}(A_1, \mathrm{wp}(A_1; \ldots A_k, R)),\ k > 1.$$

Observe that the law of the excluded miracle, wp(S, *false*) = *false*, does not necessarily hold for actions [4, 17, 18]. As an example, we have that wp(*false* $\to$ *skip*, *false*) = *true*. Also, the continuity property need not hold, as the nondeterminism may be unbounded. The other healthiness properties of [11] are valid.

3.1 Refinement of Statements and Actions

Let S and S' be sequential statements. Statement S is said to be *refined* by the statement S', denoted $S \leq S'$ if for every postcondition R,

$$\mathrm{wp}(S, R) \Rightarrow \mathrm{wp}(S', R).$$

Refinement captures the notion of statement S' preserving the correctness of statement S. More precisely, $S \leq S'$ holds if and only if $P\langle S\rangle Q \Rightarrow P\langle S'\rangle Q$ for every precondition P and postcondition Q, where $P\langle S\rangle Q$ stands for the total correctness of S w.r.t. P and Q. Hence if S is totally correct with respect to a given P and Q and $S \leq S'$, then S' will also be totally correct with respect to P and Q.

We say that the statements S and S' are (*refinement*) *equivalent*, denoted $S \equiv S'$ if $S \leq S'$ and $S' \leq S$.

The refinement relation is reflexive and transitive; i.e., it is a preorder. The statement constructors are also monotonic with respect to the refinement relation; i.e., $T \leq T' \Rightarrow S(T) \leq S(T')$ for any statement S in which T occurs as a substatement ($S = S(T)$). The refinement relation becomes a partial order if we identify statements with their associated predicate transformers.

The refinement relation provides a formalization of the stepwise refinement method for program construction. One starts with an initial high level specification/program statement S_0, and constructs a sequence of successive refinements of this, $S_0 \leq S_1 \leq \ldots \leq S_{n-1} \leq S_n$. By transitivity, the last version S_n will then be a refinement of the original program S_0. An individual refinement step may consist of replacing some substatement T of $S_i(T)$ by its refinement T'. The resulting statement $S_{i+1} = S_i(T')$ will then be a refinement of S_i by monotonicity. The refinement relation and its use in program derivation is studied in more detail in [2, 3, 17, 18].

Refinement between actions is defined in the same way as refinement between statements; i.e., an action A is *refined by* an action A', $A \leq A'$ if

$$\mathrm{wp}(A, R) \Rightarrow \mathrm{wp}(A', R)$$

for any postcondition R. The notion of total correctness can be directly extended to actions: we define $P\langle A\rangle Q$ to hold if $P \Rightarrow wp(A, Q)$ (A establishes Q when P). This is equivalent to $P \wedge gA \Rightarrow wp(bA, Q)$.

Observe that even if refinement of statements is monotonic, as stated above, action systems are *not* necessarily monotonic with respect to refinement of actions; i.e., $g_i \to S_i \leq g'_i \to S'_i$ need not imply $S_0; *[\ \ldots\ []\ g_i \to S_i \ldots] \leq S_0; *[\ \ldots\ []\ g'_i \to S'_i \ldots]$ (unless $g_i = g'_i$).

The refinement relation is quite strong, and we are therefore often faced with a situation where $S(T) \leq S(T')$ does in fact hold, but $T \leq T'$ does not hold. This means that the replacement of T by T' is correct in the specific context $S(\cdot)$ considered, even if it is not correct in every context. *Context dependent* replacements of this kind can be established correct by the following method [2, 3]: We prove

(1) $S(T) \leq S(\{Q\};T)$ (*context introduction*) and
(2) $\{Q\}; T \leq T'$ (*refinement in context*).

By monotonicity and transitivity we then have that $S(T) \leq S(T')$. The first step introduces information about the context in the form of an assert statement at the appropriate place; the second step uses this information.

Note that we are always permitted to remove any context assertion; i.e., $S[\{Q\}; T] \leq S[T]$ is always valid (because $\{Q\}; T \leq T$ is always valid).

3.2 Properties of Actions

We define below some properties of actions that will be useful in describing transformation rules for action systems, especially those by which parallelism is increased.

The way in which actions can enable and disable each other is captured by the following definitions.

$$
\begin{array}{lcl}
Q \textit{ is invariant over } A & = & Q\langle A\rangle Q, \\
A \textit{ cannot enable } B & = & \neg gB\ \langle A\rangle \neg gB, \\
A \textit{ must enable } B & = & \neg gB\ \langle A\rangle\ gB, \\
A \textit{ cannot disable } B & = & gB\ \langle A\rangle\ gB, \\
A \textit{ must disable } B & = & gB\ \langle A\rangle \neg gB, \\
A \textit{ cannot precede } B & = & \textit{true}\ \langle A\rangle \neg gB \\
A \textit{ disables itself} & = & \textit{true}\ \langle A\rangle \neg gA, \\
A \textit{ excludes } B & = & gA \Rightarrow \neg gB.
\end{array}
$$

The assertions can in all these cases be qualified to hold when some precondition Q holds initially; e.g., A cannot disable B *when* Q stands, e.g., for $Q \Rightarrow gB\ \langle A\rangle\ gB$. Q *is invariant over* $*[i : A_i]$ if Q is invariant over each A_i. The last property is generalized to sets of actions: $\{A_1, \ldots, A_m\}$ *excludes* $\{B_1, \ldots, B_n\}$ if $\vee\, g \Rightarrow \neg \vee\, gB_i$.

Another important set of properties has to do with commutativity of actions. We say that *A commutes with B* if $A; B \leq B; A$. One can show that this is the case if

(i) A cannot disable B and
(ii) B cannot enable A and
(iii) $\{gA \wedge gB\}; bA; bB \leq bB; bA$.

A sufficient condition for two actions A and B to commute is that there are no read-write conflicts for the variables that they access: none of the variables written by A is read or written by B and vice versa.

4. Stepwise Refinement of Action Systems

Because action systems are just a special kind of sequential statements, we can use the refinement calculus as such for stepwise refinement of action systems. However, the goal is to transform a more or less sequential algorithm or algorithm specification into an action system that can be executed in a highly parallel fashion, so we need special refinement rules to introduce parallelism into the execution. These do not necessarily make much sense with a strictly sequential execution (they often make the execution less efficient), but become important when considering a parallel execution.

To change a sequential statement into an action system, which is just a simple kind of loop, we need to transform the statement into a form where the iteration construct is at the outermost level, preceeded by some loop initialization. We therefore need

(i) methods for transforming sequential statements directly into iterative constructs, and
(ii) methods for combining iterative substatements into a single iterative construct.

These permit us to change parts of a sequential statement into iterative constructs, and then gradually move the iterations outwards through the statement until we get an action system.

The above rules do not necessarily introduce any real parallelism in the resulting action system. For this we also need

(iii) methods for changing variables in substatements.

These will make it possible to change the way in which the program state is represented by variables in the system, in order to replace a centralized representation by a distributed one. Typically we replace a single variable by a number of variables, one for each process. These new variables hold the same information as the original variable, but permit the information to be accessed in a distributed fashion. This makes the processes less dependent on each other, so that more activity can go on in parallel.

We also need general rules and methods for massaging a program into a form where the above methods can be applied, and also for improving the efficiency of the program in general. These are not specific to the construction of action systems but apply to any kind of statements. Especially important are rules by which we can assert something about the context of substatements to be refined and general rules for changing the control structure of statements. We do not consider these here in any more detail. Rules for context introduction are described in detail in [3], while the rules for changing control structures are often straightforward and usually easy to check.

4.1 Constructing and Merging Loops

The first three rules tell us when a statement can be directly transformed into a loop:

Rule 1 (*Making a loop directly*)

$$\{Q\}; S \le *[b \to S]$$

provided that

(i) $Q \Rightarrow b$ and
(ii) $b \to S$ disables itself.

Thus, a sufficient condition for directly turning a statement into a loop is that the guard holds initially and is disabled by the statement itself. This guarantees that the statement is executed exactly once in the loop.

Rule 2 (Changing a sequence to a loop)

$$\{Q\}; (i :1..n :S_i) \le * [i :1..n : b_i \to S_i]$$

provided that

(i) $Q \Rightarrow b_1$,
(ii) $b_i \to S_i$ enables $b_{i+1} \to S_{i+1}$, for $i = 1, \dots, n-1$,
(iii) $b_n \to S_n$ establishes $\neg \vee b_i$ and
(iv) $b_i \to S_i$ excludes all other actions $b_j \to S_j$, $j \ne i$.

This rule will change a sequence to a loop. However, exactly the same statements are executed as before and in the same order. In particular, there is no parallelism in the execution of the statements, as they exclude each other. Observe that Rule 1 is a special case of Rule 2 ($n = 1$).

Rule 3 (Changing a conditional to a loop)

$$\{Q\}; [i :1..n : b_i \to S_i] \le *[i :1..n : b_i \to S_i]$$

provided that

(i) $b_i \to S_i$ establishes $\neg \vee b_i$ when Q, for every $i = 1, \dots, n$.

Changing a conditional into a loop is thus always permitted if every action disables all the actions of the conditional statement.

Rule 4 (Merging a sequence of loops)

$$*[i : A_i]; *[j : B_j] \le *[i : A_i \ [] \ j : B_j]$$

provided that

(i) for every i, j, B_j cannot enable or disable A_i,
(ii) for every i, j, either B_i cannot precede A_j or A_j commutes with B_i and
(iii) $*[j : B_j]$ terminates when $\vee$ g A_i.

This rule does introduce more parallelism in the execution of an action system. We have initially a situation where the execution of two loops must be done in sequence: the actions of the second loop may not start before all the actions of the first loop have become disabled. The rule permits the execution of the actions from the two loops to be overlapped, provided the essential sequential constraints are preserved.

Rule 5 (Merging a conditional composition of loops)

$$\{Q\}; [b_1 \to S_1;*[i : A_i] \ [] \ b_2 \to S_2; *[j : B_j]]$$
$$\le *[b_1 \to S_1 \ [] \ b_2 \to S_2 \ [] \ i : A_i \ [] \ j : B_j]$$

provided that

(i) $b_1 \to S_1$ and $b_2 \to S_2$ exclude $\{i : A_i\}$ and $\{j :B_j\}$,
(ii) $b_1 \to S_1$ and $\{i : A_i\}$ cannot enable $b_1 \to S_1$ or $b_2 \to S_2$ or $\{j : B_j\}$, and
(iii) $b_2 \to S_2$ and $\{j : B_j\}$ cannot enable $b_1 \to S_1$ or $b_2 \to S_2$ or $\{i :A_i\}$.

This rule permits us to create a single action system from a conditional composition of action systems. The replacement can be made provided that the actual execution of the systems preserves the original distinction between the two action systems.

Rule 6 (Merging nested loops)

$$\{Q\}; *[b_0 \to S_0; *[i : 1..n : A_i] \ [] \ j : B_j]$$
$$\le *[b_0 \to S_0 \ [] \ i : 1..n : A_i \ [] \ j :B_j]$$

provided that

(i) $Q \Rightarrow \neg(\vee\, i : 1..n : gA_i)$,
(ii) $\{j : B_j\}$ cannot enable or disable $\{i : 1..n : A_i\}$,
(iii) $A_0 = b_0 \to S_0$ is excluded by $\{i : 1..n : A_i\}$ and
(iv) the actions in $\{j : B_j\}$ that are not excluded by $\{i : 0..n : A_i\}$ can be partitioned into left movers $\{k : L_k\}$ and right movers $\{h : R_h\}$, such that
  (a) for each $i = 0, \dots, n$ and k, either A_i cannot precede L_k or L_k commutes with A_i,
  (b) for each $i = 0, \dots, n$ and h, either R_h cannot precede A_i or A_i commutes with R_k,
  (c) for each k and h, either R_h cannot precede L_k or L_k commutes with R_h and

(d) $*[h : R_h]$ terminates when $(\vee\, i : 0..n : gA_i)$.

The last rule shows us how to refine the atomicity of an action. Initially we have a loop where one of the action bodies is in fact an action system. However, only the outermost actions can be executed in parallel, so the inner action system is executed sequentially, as one big action. The rule shows us under what conditions we are permitted to merge the two loops into a single loop, where the inner actions are executed interleaved with the actions of the outer loop. This means that the inner action system is not executed atomically anymore, but that its execution may overlap with execution of actions from the outer loop.

The above rules are all stated without considering possible invariants that may hold for the actions involved. These invariants are very important in practice, because many reasonable refinements turn out to hold only because some specific invariant holds. However, we can always take these invariants into account by, e.g., temporarily adding them to the guards, proving the required conditions with these strengthened guards, and then later removing the invariants again from the guards.

The following rule can be used to change the guards of an iteration statement.

Rule 7 (*Changing guards in loops*)
$\{Q\}; *[i : b_i \rightarrow S_i] \equiv \{Q\}; *[i : b'_i \rightarrow S_i]$

provided that

(i) $Q \wedge b_i \Leftrightarrow Q \wedge b'_i$ for every i and
(ii) Q is invariant of $*\ [i : b_i \rightarrow S_i]$.

In particular, we have that

$$\{Q\}; *[i : b_i \rightarrow S_i] \equiv \{Q\}; *[i : b_i \wedge I \rightarrow \{b_i \wedge I\}; S_i]$$

when I is an invariant of the loop, so we may add the invariant to each of the guards. Furthermore, the invariant and the guard may be added as a context assertion for the body of the action. In the other direction, we may remove a common conjunct from all the guards if we can show that this conjunct is, in fact, an invariant of the loop. (A context assertion, again, may always be removed.)

The methods presented in this subsection are studied in detail in [4] and [19].

4.2 Changing Variables

The other important class of refinement rules for parallelization involve changing the way in which the program state is represented by variables. We need this, e.g., to change a scalar variable into an array for loop parallelization. This again models the replication of a variable across a collection of processes, so that each process can work with its own copy of the variable, with readings and updates taking place in parallel when possible.

We can add assignment statements to a new variable x anywhere in a statement S if x is made into a local variable:

Rule 8 (Adding auxiliary variables)

$$S \equiv [var\ x; S[x := h_1/skip^1, \dots , x := h_n/skip^n]]$$

provided that

(i) S does not contain any occurrence of x.

Here $x := h/skip$ denotes the substitution of $x := h$ for *skip* and $skip^1, \dots , skip^n$ denote different occurrences of the *skip* statement in S. This refinement equivalence can be proved correct in the weakest precondition calculus by structural induction.

Assume that we initially have a statement $[var\ x; S]$ and we want to replace the variables x with some other variables y, changing S to S' accordingly, such that

$$[var\ x; S] \le [var\ y; S'].$$

(In essence, we are changing the data representation in this statement). We can achieve this by the following sequence of steps [8]:

(1) *Introduce new variables* y. Let $S_1 = S[y_1 := e_1/skip^1, \dots , y_n := e_n/skip^n]$. Applying Rule 8 and monotonicity of refinement, we have

$$[var\ x; S] \equiv [var\ x; [var\ y; S_1]].$$

(2) *Introduce context assertion relating* x *and* y. Let $Q_1, \dots , Q_k$ be assertions on x and y, and assume that context introduction gives us

$$S_1 \le S_1[\{Q_1\}\ ; T_1/T_1, \dots , \{Q_k\}\ ; T_k/T_k],$$

where $T_1, \dots , T_k$ are substatements of S_1 that refer to the variables in x.

(3) *Replace statements on* x *with statements on* y. Let $T'_1, \dots , T'_k$ be statements that do not refer to variables in x, and assume that we can show that

$$\{Q_1\}\ ; T_1 \le T'_1$$
$$\vdots$$
$$\{Q_k\}\ ; T_k \le T'_k.$$

By transitivity of refinement, we now have

$$S_1 \leq S_1[T'_1/T_1, \dots, T'_k/T_k] = S_2$$

(4) *Remove old variables x.* Assume that all the remaining occurrences of x are now in assignments to variables in x, i.e.,

$$S_2 = S'_2[x := h_1/skip^1, \dots, x := h_m/skip^m]$$

where S'_2 does not contain any occurrences of x.
Applying Rule 8 again then gives us

$$\begin{aligned}[var\ x; [var\ y; S_1]] &\leq [var\ x; [var\ y; S_2]] \\ &\equiv [var\ y; [var\ x; S_2]] \\ &\equiv [var\ y; [var\ x; S'_2\ [x := h/skip^1, \dots, x := h_m/skip^m]]] \\ &\equiv [var\ y; S'_2].\end{aligned}$$

Transitivity of refinement then gives us the desired result

$$[var\ x; S] \leq [var\ y; S'_2].$$

5. Example Derivation: Matrix Multiplication

In this section we show the main refinement steps in the construction of a parallel and distributed algorithm for matrix multiplication.

We are interested in performing the multiplication

$$C = A \cdot B$$

where C, A and B are $N \times N$ matrices. We assume that the target architecture consists of N processors configured in a ring. Each processor has a local memory and there is no global memory. Processor i can participate in two-process shared actions with its two neighbours. The intention is that process i should store row i from A, B and C, and that the updating of the rows in C could proceed in parallel. Our task is to design a decentralized and locally scheduled action system of degree 2 for matrix multiplication, to be implemented on this architecture.

Let us start by deriving an alternative definition of matrix multiplication that is more suitable for parallelization. For $i, j = 1, \dots, N$, we have

$$C_{i,j} = \sum_{k=1}^{N} A_{i,k} \cdot B_{k,j}$$

$$= \sum_{k=1}^{i-1} A_{i,k} \cdot B_{k,j} + \sum_{k=i}^{N} A_{i,k} \cdot B_{k,j}$$

$$= \sum_{k=1}^{N} A_{i,k} \cdot B_{k,j} + \sum_{k=i}^{i-1} A_{i,k} \cdot B_{k,j}$$

$$= \sum_{k'=0}^{N-i} A_{i,i+k'} \cdot B_{i+k',j} + \sum_{k'=N-i+1}^{N-1} A_{i,i+k'-N} \cdot B_{i+k'-N,j}$$

$$= \sum_{k'=0}^{N-i} A_{i,r(i,k')} \cdot B_{r(i,k'),j} + \sum_{k'=N-i+1}^{N-1} A_{i,r(i,k')} \cdot B_{r(i,k'),j}$$

where $r(i, k') = (i + k' - 1)\ modN + 1$

$$= \sum_{k'=0}^{N-i} A_{i,r(i,k')} \cdot B_{r(i,k'),j}$$

Hence, we can compute the value of $C_{i,j}$ by

*C.i.j:=0; (k: 0..N–1: C.i.j:= C.i.j+ A.i.r(i,k) * B.r(i,k).j)*

We have essentially changed the order in which the columns of the B matrix are referenced: the computation of the value for an element $C.i.j$ now starts with that particular element of column j of the B matrix which is located in processor i instead of the element located in processor 1.

This gives us our initial algorithm for matrix multiplication:

```
[function r(i, k) = (i + k – 1) mod N + 1;          (S0)
(i:1..N:(j:1..N: C.i.j := 0 ));
(k: 0..N – 1:
    (i:1..N: (j:1..N: C.i.j := C.i.j+ A.i.r(i,k)
                                   * B.r(i,k).j))))]
```

We compute here all elements $C.i.j$ at the same time for a specific $k = 0, \dots, N - 1$. This is permitted because the computations of the different elements $C.i.j$ are independent of each other. (The index calculations would be slightly simpler if the array indexing was from $0, \dots, N - 1$, but our choice does not complicate the successive derivations, so we keep it this way).

We will in the sequel usually not explicitly state that a specific program version is a refinement of another version. Instead, we adopt the convention that successively numbered versions are refinements; i.e., $Si \leq S(i + 1)$. The same holds for versions on a nested level; i.e., $Si.j \leq Si.(j + 1)$ and so on.

5.1 Shifting the B Matrix

Our first refinements are intended to localize the matrix multiplications that have to be done, so that each multiplication only concerns elements with the same row index. In the final architecture all elements with the same row index will be stored in the same processor, so that each multiplication becomes an internal action of a processor.

We will achieve this by rotating the B matrix cyclically after each iteration of k. The transformations below are an example of the method for changing variables described in the previous section. When describing operations on the element of the matrices, we often refer to the entire matrix or to a row of a matrix. Let us first add an auxiliary matrix B' to the program, together with assignments to it:

```
|[function r(i, k) = (i + k – 1) mod N + 1;                (S1)
var B'.i.j is integer, i: 1..N, j: 1..N;
C:= 0; B' := B;
(k: 0..N – 1: {Q1};
     (i:1..N: (j :1..N: C.i.j:= C.i.j + A.i.r(i,k)
                              * B.r(i,k).j)); « S.1.1 »
     ||i:1..N:B'.i:= B'.r(i,1) ||); {Q2} ]
```

We have $S0 \leq S1$, by Rule 8. We use above the notation « $S1.1$ » to name a substatement in the program (it refers to the immediately preceeding substatement plus the eventual context assertion). Similarly, when writing a context assertion in the program, we say that the context assertion in question can be inserted at the place indicated.

We can show that $Q1$ holds at the place indicated,

$$Q1 = (\forall\, i{:}1..N{:}\ (\forall\, j{:}1..N{:}\ B'.i.j = B.r(i,k).j)).$$

Hence, we may use B' instead of B for updating C:

```
(i: 1..N: (j: 1..N: C.i.j:= C.i.j + A.i.r(i,k)*B'.i.j))
                                                   (S1.2)
```

We can also show that upon termination the B' matrix has been restored to B, i.e.,

$$Q2 = (B' = B)$$

holds. We may therefore use B directly in the computation, instead of B' and remove B' :

```
|[function r(i,k) = (i + k – 1) mod N + 1;                 (S2)
C:= 0;
(k: 0..N – 1:
     (i:1..N: (j:1..N: C.i.j:= C.i.j + A.i.r(i,k) * B.i.j));
     ||i:1..N: B.i:= B.r(i,1) || « S2.1 »
)]
```

Next we replace the parallel shift $S2.1$ of the B-matrix by a sequential shift:

```
|[function r(i,k) = (i + k – 1) mod N + 1;                 (S3)
function q(i) = (i + 1) mod (N + 1);
var B.0.j is integer, j: 1..N;
C:= 0;
(k: 0..N – 1:
     (i:1..N: (j:1..N: C.i.j:= C.i.j + A.i.r(i,k) * B.i.j));
     (i:0..N: (j:1..N: B.i := B.q(i))) « S3.1 »
)]
```

The sequential shift $S3.1$ uses the auxiliary function $q(i) = (i + 1)$ *mod* $(N + 1)$. We also needed to introduce new local variabels $B.0.j$, $j = 1, \ldots, N$ as temporary storage for the cyclic shifting.

Introducing some abbreviations for future convenience, we then have the following version of matrix multiplication :

```
|[function r(i,k) = (i + k – 1) mod N + 1;                 (S4)
function q(i) = (i + 1) mod (N + 1);
var B.0.j is integer, j: 1..N;
procedure zeroC = (i:1..N: (j :1..N: C.i.j:= 0));
procedure mult.i.k =
     (j:1..N: C.i.j:= C.i.j + A.i.r(i,k) * B.i.j),
     i :1..N, k :0..N – 1;
procedure shift.i = (j:1..N: B.i.j:= B.q(i).j), i: 0..N;
zeroC;
(k: 0..N – 1 :
     (i:1..N: mult.i.k); (i:0..N: shift.i)) « S4.1 » ]
```

The notation indicates that we have, e.g., $N + 1$ different versions of the shift procedure, *shift*.0, ... , *shift*.N.

5.2 Parallelizing the Multiplications within a Cycle

The multiplications within each cycle are now done in a strictly sequential manner, although they affect disjoint rows. Our next task is to parallelize these. In the sequel we will focus on the main loop $S4.1$, taking this as the basis for further refinements:

```
(k: 0..N – 1:                                              (S4.1)
     (i:1..N: mult.i.k);
     (i:0..N: shift.i))
```

Let us first move the first shifting operation, *shift*.0, to precede the multiplications:

```
(k:0..N – 1:                                               (S4.2)
     shift.0;
     (i: 1..N: mult.i.k); « S4.2.1 »
     (i: 1..N: shift.i))
```

The operation *shift*.0 commutes with each *mult.i.k* operation because there are no read-write conflicts: the *shift*.0 opeation reads variables $B.1$ and writes variables $B.0$ while the *mult.i.k* operation reads $C.i, A.i.k, B.i, i = 1, \dots, N$ and writes $C.i, i = 1, \dots, N$. This refinement is carried out because the operation *shift*.0 differs from the other shift operations.

We add new variables *h.i* to S4.2.1, to keep track of which multiplications already have been done:

```
[var h.i is boolean, i:1..N;                          (S4.2.2)
(i: 1..N: h.i:= true);
(i: 1..N: mult.i.k; h.i:= false) « S4.2.2.1 »];
```

Next we change the first two multiplications in S4.2.2.1 into loops:

```
* [h.1 → mult.1.k; h.1:= false];                      (S4.2.2.2)
* [h.2 → mult.2.k; h.2:= false];
(i: 3..N: mult.i.k; h.i:= false);
```

These are permitted by Rule 1, because the actions are initially enabled ($h.1 = h.2 = true$ initially) and they disable themselves.

We can now merge the two loops into one:

```
* [h.1 → mult.1.k; h.1:= false                        (S4.2.2.3)
 [] h.2 → mult.2.k; h.2:= false];
(i: 3..N: mult.i.k; h.i:= false);
```

The merge is permitted by Rule 4: (1) *mult*.2.*k* cannot disable or enable *mult*.1.*k*, (2) *mult*.1.*k* and *mult*.2.*k* commute (because they affect different rows) and (3) the second loop must terminate.

The other multiplications can then also be merged into this loop, one by one, by the same reasoning. This gives us:

```
* [i: 1..N: h.i → mult.i.k; h.i:= false];             (S4.2.2.4)
```

Hence we have derived the following refinement of statement S4.2:

```
(k:0..N – 1:                                          (S4.3)
      shift.0;
      [var h.i is boolean, i:1..N;
      (i: 1..N: h.i:= true);
      * [i: 1..N: h.i → mult.i.k; h.i:= false]];
      (i: 1..N: shift.i))
```

All multiplications within a cycle can now be done in parallel.

5.3 Merging Shift Operations with Multiplications

Our next task is to move the shift operations into the multiplication loop. We are trying to make the whole cycle into a single action system, so the sequential composition of multiplications and shifts must be removed from the cycle.

We first add new variables *m.i* and assignments to these to keep track of which shift operations already have been done:

```
[var h.i is boolean,:1..N; var m.i is boolean, i:0..N;
                                                      (S4.4)
(k:0..N – 1:
      shift.0; (i: 1..N: h.i:= true); (i: 0..N: m.i:= false);
      m.1:= true;
      * [i: 1..N: h.i → mult.i.k; h.i:= false]]; {Q3};
      (i: 1..N: shift.i; m.i,m.q(i):= false, true )
)]
```

We have also moved the declarations outwards to the beginning of the block. Assertion $Q3$ holds after the multiplication loop,

$$Q3 = (\forall\ i\!: 1..N\!: \neg h.i) \wedge m.1.$$

We then make *shift*.1 into a loop:

```
[var h.i is boolean, i:1..N; var m.i is boolean, i:0..N;
                                                      (S4.5)
(k:0..N – 1:
      shift.0; (i: 1..N: h.i:= true); (i: 0..N: m.i:= false);
      m.1:= true;
      * [i: 1..N: h.i → mult.i.k; h.i:= false];
      * [m.1 ∧ ¬h.1 → shift.1; m.1, m.2:= false, true];
      (i: 2..N: shift.i; m.i,m.q(i):= false, true)
)]
```

This is permitted because (1) the action disables itself and (2) the action is initially enabled by assertion $Q3$ and the initialization of $m.1$.

The *shift*.1 operation is then merged with the preceeding loop:

```
[var h.i is boolean, i:1..N; var m.i is boolean, i:0..N;
                                                      (S4.6)
(k:0..N – 1:
      shift.0; (i: 1..N: h.i:= true); (i: 0..N: m.i:= false);
      m.1:= true;
      * [i: 1..N: h.i → mult.i.k; h.i:= false
       [] m.1 ∧ ¬h.1 → shift.1; m.1, m.2:= false, true];
      (i: 2..N: shift.i; m.i,m.q(i):= false, true )
)]
```

This is permitted by Rule 4 because

1. *shift*.1 does not disable or enable any multiplication,
2. *shift*.1 commutes with every *mult.i.k*, $i \neq 1$, while for $i = 1$, *shift*.1 cannot precede *mult*.1.*k*, as

$$m.1 \wedge \neg h.1 \Rightarrow wp(shift.1;\ m.1, m.2{:=}\ false, true, \neg h.1) = \neg h.1$$

and

3. the *shift*.1 loop terminates.

The rest of the *shift.i* actions can then also be merged one by one into the multiplication loop:

[var h.i is boolean, i:1..N; var m.i is boolean, i:0..N; *(S4.7)*
(k:0..N – 1:
 shift.0; (i: 1..N: h.i:= true); (i: 0..N: m.i:= false);
 m.1:= true;
 * *[i: 1..N: h.i → mult.i.k; h.i:= false*
 [] i:1..N: m.i ∧ ¬h.i → shift.i; m.i,m.q(i):= false, true];
)]

The *shift.i* operations do not interfere with the multiplication actions, for $i = 1, \dots, N$, by the same reasoning as above. We also need to show that *shift.i* does not interfere with the operations *shift*.1, ... , *shift*.$(i-1)$ already added to the loop. This is seen as follows:

1. *shift.i* does not enable or disable *shift.j*, $1 \leq j < i$.
2. *shift.i* cannot precede *shift.j*, $j < i$, because

$$m.i \wedge \neg h.i \Rightarrow wp(shift.i;\ m.i, m.q(i){:=}\ false, true, \neg m.j \vee h.j) = \neg m.j \vee h.j$$

 This again is valid because

$$Q4 = (\forall\ i : 0..N: m.i \Rightarrow (\forall\ j{:}0..N: j \neq i \Rightarrow\ \neg m.j))$$

is invariant of all the actions in the loop where *shift*.1, ... , *shift*.$(i-1)$ have been inserted, and is also an invariant of the action with *shift.i*.

The variables *m.i* simulate here a token that is passed from one process to the next in the ring to guarantee that the shift operations are done in the correct order.

5.4 Move Cycle Initialization Inside Loop

Next we make the initialization of each cycle into an action of the multiplication loop. We first add variables *iter.i*, to keep count of the number of iterations already done and to prevent successive cycles from being mixed up:

[var h.i is boolean, i:1..N; var m.i is boolean, i:0..N; *(S4.8)*
var iter.i is integer, i:0..N;
(i: 0..N: iter.i:= 0);
(k:0..N – 1: {Q5};
 shift.0; (i: 1..N: h.i, m.i:= true, false);
 m.0, m.1:= false,true; iter.0:= iter.0+1; {Q6};
 * *[i: 1..N: h.i → mult.i.k; h.i:= false*
 [] i: 1..N: m.i ∧ ¬h.i → shift.i;
 m.i,m.q(i),iter.i:= false, true, iter.i + 1]
)]

Assertion $Q5$ holds at the indicated place,

$$Q5 = (\forall\ i{:}0..N: iter.i = k).$$

Thus the iteration counters are in step whenever the next cycle is about to start.

Assertion $Q6$ is an invariant of the loop,

$$Q6 = (\forall i{:}\ 1..N: h.i \Rightarrow iter.i = k) \wedge (\forall i{:}\ 1..N: m.i \Rightarrow (\forall j{:}\ i..N: iter.i = k) \wedge (\forall j{:}\ 0..i-1: iter.j = k+1))$$

Because of it, we can add the assertion $iter.i = k$ to the guards of the loop by Rule 7:

[var h.i is boolean, i:1..N; var m.i is boolean, i:0..N; *(S4.9)*
var iter.i is integer, i:0..N;
(i: 0..N: iter.i:= 0);
(k:0..N – 1:
 shift.0; (i: 1..N: h.i, m.i:= true, false);
 m.0, m.1:= false,true; iter.0:= iter.0+1;
 * *[i: 1..N: h.i* ∧ iter.i = k → *mult.i.k; h.i: = false*
 [] i: 1..N: m.i ∧ ¬h.i ∧ iter.i = k →
 shift.i; m.i,m.q(i),iter.i:= false, true,iter.i+1]
)]

We can then set the variables *h.i* to true in the *shift.i* actions and also replace the reference to *k* in *mult.i.k* by *iter.i*:

[var h.i is boolean, i:1..N; var m.i is boolean, i:0..N; *(S4.10)*
var iter.i is integer, i:0..N;
(i: 0..N: iter.i:= 0);
(k:0..N – 1:
 shift.0; (i: 1..N: h.i, m.i:= true, false);
 m.0, m.1:= false,true; iter.0:= iter.0+1;
 * [i: 1..N: h.i ∧ iter.i = k → mult.i.(iter.i);
 h.i:= false

```
    [] i: 1..N: m.i ∧ ¬h.i ∧ iter.i = k →
      shift.i; m.i, m.q(i),iter.i,h.i:=
                    false, true, iter.i + 1,true ];{Q7}
)|
```

The change of k to *iter.i* is justified by the guard. The updating of *h.i* is justified because *iter.i* is set to $k + 1$ at the same time, so the multiplication actions do not become enabled anew in this cycle.

Assertion $Q7$ will now hold at the indicated place,

$$Q7 = (\forall\ i{:}1..N{:}\ h.i \wedge \neg m.i) \wedge m.0$$

Hence we can remove the initializations between the cycles as unnecessary. The initialization of the first cycle is moved to the beginning of the program. We add initial assignments to $m.0$ and $m.1$ to get into step with the situation after the next iteration:

```
[var h.i is boolean, i:1..N; var m.i is boolean, i:0..N;
                                                (S4.11)
var iter.i is integer, i:0..N;
(i: 0..N: iter.i:= 0); (i: 1..N: h.i, m.i:= true,false);
m.0:= true;
(k:0..N – 1:
      (shift.0; m.0, m.1:=false,true; iter.0:= iter.0 + 1
                                              « S4.11.1 »;
      * [i: 1..N: h.i ∧ iter.i = k → mult.i.(iter.i);
                                    h.i:=false
        [] i: 1..N: m.i ∧ ¬h.i ∧ iter.i = k →
            shift.i; m.i,m.q(i),iter.i,h.i:=false,
                                    true,iter.i+1,true ]
)|
```

The statement $S4.11.1$ can now be changed into a loop

```
* [ m.0 ∧ iter.0 = k →                      (S4.11.2)
      shift.0; m.0, m.1:=false,true; iter.0: = iter.0 + 1]
```

and merged with the loop following it, to get:

```
[var h.i is boolean, i:1..N; var m.i is boolean, i:0..N;
                                                (S4.12)
var iter.i is integer, i:0..N;
(i: 0..N: iter.i:= 0); (i: 1..N: h.i, m.i:= true, false);
m.0:= true;
(k:0..N – 1:
      * [ m.0 ∧ iter.0 =k →
          shift.0; m.0, m.1:=false,true;
          iter.0:= iter.0 + 1
        [] i: 1..N: h.i ∧ iter.i = k → mult.i.(iter.i);
                                      h.i:=false
        [] i: 1..N: m.i ∧ ¬h.i ∧ iter.i = k →
            shift.i; m.i,m.q(i),iter.i,h.i:=
                        false, true, iter.i + 1,true ]
)|
```

This is again justified by Rule 4. We omit the details of the proof here.

5.5 Change Outer Iteration into Loop

We are now ready for the final steps, where we change the whole sequence of cycles into a single action system. We can again use Rule 4, merging all actions into a single loop in the same way as was done above. This gives us

```
[var h.i is boolean, i:1..N; var m.i is boolean, i:0..N;
                                                (S4.13)
var iter.i is integer, i:0..N;
(i: 0..N: iter.i:= 0); (i: 1..N: h.i, m.i:= true,false);
m.0:= true;
* [ k:0..N – 1: m.0 ∧ iter.0 =k →
          shift.0; m.0, m.1:= false, true;
          iter.0:= iter.0 + 1
  [] k:0..N – 1, i: 1..N: h.i ∧ iter.i=k → mult.i.(iter.i);
                                          h.i:=false
  [] k:0..N – 1, i: 1..N: m.i ∧ ¬h.i ∧ iter.i = k →
          shift.i; m.i, m.q(i), iter.i,h.i:=
                false, true, iter.i + 1,true ]
)|
```

The justification of this step is similar to the preceding ones (although somewhat tedious).

We can now combine all those actions that have the same body into a single action with this same body, but with the disjunction of all the action guards as its guard:

```
[var h.i is boolean, i:0..N; var m.i is boolean, i:0..N;
                                                (S4.14 )
var iter.i is integer, i:0..N;
(i: 1..N: iter.i, h.i, m.i:= 0,true, false);
      iter.0, h.0, m.0:= 0,false,true;
* [ i: 1..N: h.i ∧ iter.i< N → mult.i.(iter.i); h.i:= false
  [] i: 0..N: m.i ∧ ¬h.i ∧ iter.i < N →
                  shift.i; m.i,m.q(i), iter.i,h.i:=
                        false, true, iter.i + 1,i ≠ 0 ]
]
```

This follows from the fact that

$$(\vee\ k{:}\ 0..N-1{:}\ m.0 \wedge iter.0 = k) \Leftrightarrow m.0 \wedge iter.0 < N$$

(in combination with the loop invariant $iter.i \geq 0$) and similarly for the other actions. We have also massaged the shift actions a little bit so that they also all have the same body.

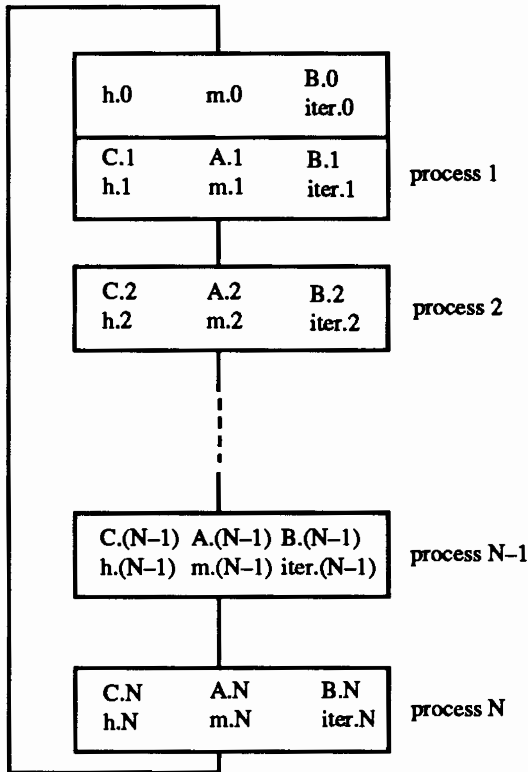

Figure 2. The structure of the parallel program.

5.6 Final Version

The final version of our algorithm is now as follows:

```
[function r(i,k) = (i + k – 1) mod N + 1;                    (S5)
function q(i) = (i + 1) mod (N + 1);
var B.0.j is integer, j: 1..N;
var h.i is boolean, m.i is boolean,
    iter.i is integer, i:0..N;
procedure zeroC = (i:1..N:(j:1..N: C.i.j:= 0));
procedure mult.i = (j:1..N: C.i.j:= C.i.j +
                        A.i.r(i,iter.i) * B.i.j), i:1..N;
procedure shift.i = (j:1..N: B.i.j:= B.q(i).j), i:0..N;
zeroC;
(i: 1..N: iter.i, h.i, m.i:= 0,true,false);
    iter.0, h.0, m.0:= 0,false, true;
* [ i: 1..N: h.i ∧ iter.i< N → mult.i; h.i:=false
  [] i: 0..N: m.i ∧ ¬h.i ∧ iter.i < N →
              shift.i; m.i,m.q(i),iter.i,h.i:=
                        false, true, iter.i + 1,i ≠ 0 ]
]
```

We can implement this program on either $N + 1$ or N processors, depending on whether we want to allocate a separate processor to the variables with index 0. In both cases the resulting program is obviously decentralized and has only one and two-process actions, the latter involving only neighbours in the ring. The only action with two processes is *shift.i* for $i \geq 1$. This action will involve processes *p.i* and *p.*$(i + 1)$. If we choose *p.i* to be a committer in this action, then no other action in which *p.i* is involved can be enabled at the same time as *shift.i* is enabled. Hence, the system is locally scheduled, as required.

The process structure is shown in Figure 2. All multiplications can be done in parallel in the program. The shift operations can proceed simultaneously with the multiplications, moving the m-token around the processor ring, but only one *shift.i* operation may be in progress at a time. Multiplications from two successive cycles can be in progress at the same time. The only synchronization is that the next row of the B matrix may not be shifted into a processor before the previous row has been used for the multiplication.

6. Concluding Remarks

The refinement calculus and the action system formalism has been combined into a formal method for the stepwise construction of parallel algorithms. The combination gives us a methodology where the entire derivation of such systems can be done in the context of purely sequential programs.

In [8] a detailed case study of the method was carried out: a parallel and distributed algorithm for solving linear systems of equations was constructed by stepwise refinement, starting from the original sequential Gaussian elimination algorithm. The study showed that the main body of the derivation could be done using only a few transformation rules and methods by which the parallelism in the algorithm was increased step by step. In this derivation the atomicity refinement rule (Rule 6) played an equally central role as the rule for merging loops has played in the derivation of the matrix algorithm above.

Stepwise refinement of parallel programs is also studied in [10] with the UNITY approach. UNITY programs are very similar to action systems, although their actions are restricted to conditional, deterministic assignment statements. The approach to refinement is also different from ours: the specification of the program is refined instead of refining the program text as we do here. Another approach to the construction of totally correct parallel programs is presented in [14]. Commutativity as a tool in verifying correctness of parallel programs is also emphasized in [15] and [16].

Acknowledgements: The work reported here was supported by the Academy of Finland, the Ministry of Education in Finland and the FINSOFT III program sponsored by the Technology Development Centre of Finland. We would like to thank Jochum von Wright for helpful discussions on the topics treated here.

References

1. Back RJR (1978) On the Correctness of Refinement in Program Development. Ph.D. thesis, Report A-1978-4, Department of Computer Science, University of Helsinki
2. Back RJR (1980) Correctness Preserving Program Refinements: Proof Theory and Applications. Mathematical Center Tracts 131, Mathematical Centre, Amsterdam 1980
3. Back RJR (1988) A Calculus of Refinements for Program Derivations. Acta Informatica, Vol. 25, Springer-Verlag, 593–624
4. Back RJR (June 1989) Refining Atomicity in Parallel Algorithms. Åbo Akademi, Reports on computer science and mathematics, Ser. A, No, 57, 1988. To appear in PARLE Conference on Parallel Architectures and Languages Europe, Eindhoven, the Netherlands, June 12–16, 1989
5. Back RJR, Hartikainen E, Kurki-Suonio R (1985) Multi-process Handshaking on Broadcasting Networks. Åbo Akademi, Reports on Computer Science and Mathematics, Ser. A, No, 42
6. Back RJR, Kurki-Suonio R (August 1983) Decentralization of Process Nets with Centralized Contro. 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing, Montreal, Canada, 131–142
7. Back RJR, Kurki-Suonio R (October 1988) Distributed Co-Operation with Action Systems. ACM Transactions on Programming Languages and Systems 10, 513–554
8. Back RJR, Sere K (1990) Stepwise Refinement of Parallel Algorithms. Science of Computer Programming 13, North-Holland, 133–180
9. Back RJR, Sere K (January 1990) Deriving an Occam Implementation of Action Systems. In Proc. of the 3rd Refinement Workshop BCS FACS/IBM UK Laboratories, Oxford University, Hursley Park, England, to appear
10. Chandy KM, Misra J (1988) Parallel Program Design: A Foundation. Addison-Wesley
11. Dijkstra EW (1976) A Discipline of Programming. Prentice-Hall International
12. Hoare CAR (August 1978) Communicating Sequential Processes. CACM, Vol. 21, No, 8, 666–677
13. INMOS Ltd. (1984) occam Programming Manua. Prentice-Hall International
14. Van Lamsweerde A, Sintzoff M (1979) Formal Derivation of Strongly Correct Concurrent Programs. Acta Informatica, Vol. 12, 1–31, Springer-Verlag
15. Lengauer C (1982) A Methodology for Programming with Concurrency: The Formalism. Science of Computer Programming 2, North-Holland, 19–52
16. Lipton RJR (December 1975) Reduction: A Method of Proving Properties of Parallel Programs. CACM, Vol. 18, No, 12, 717–721
17. Morgan C (July 1988) The Specification Statement. ACM Transactions on Programming Languages and Systems, Vol. 10, No, 3, 403–419
18. Morris JM (1987) A Theoretical Basis for Stepwise Refinement and the Programming Calculus. Science of Computer Programming 9, North-Holland, 287–306
19. Sere K (1990) Stepwise Derivation of Parallel Algorithms. Ph.D. Thesis, Åbo Akademi, Department of Computer Science
20. Von Wright J (manuscript) A Derivation of a Parallel Matrix Multiplication Algorithm. Åbo Akademi